Theoretical Computer Science 412 (2011) 4297–4311

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Fault containment in weakly stabilizing systems

Anurag Dasgupta, Sukumar Ghosh *, Xin Xiao

Department of Computer Science, The University of Iowa, United States

ARTICLE INFO

Keywords:
Fault containment
Weak stabilization
Persistent bit
Leader election
Fault gap

ABSTRACT

Research on fine tuning stabilization properties has received attention for more than a decade. This paper presents probabilistic algorithms for fault containment. We demonstrate two exercises in fault containment in a *weakly stabilizing system*, which expedite recovery from single failures, and confine the effect of any single fault to the constant-distance neighborhood of the faulty process. The most significant aspect of the algorithms is that the *fault gap*, defined as the smallest interval after which the system is ready to handle the next single fault with the same efficiency, is independent of the network size. We argue that a small fault gap increases the availability of the fault-free system.

1. Introduction

A distributed system is *weakly stabilizing* [14,6], when its *legitimate configuration* is reachable from any starting configuration, and the legitimate configuration is closed under the given system actions. Arbitrary configurations may be caused by transient failures that can corrupt the system state. Due to the weakening of the stabilization property, it is not possible to bound the stabilization time under a deterministic scheduler, and randomized scheduling becomes necessary to guarantee eventual recovery. Furthermore, in well-designed systems, and with improved reliability of the system components, the possibility of a massive failure is quite low, and single failures are much more likely to occur. To increase the efficiency of fault tolerance, it is important to guarantee a fast recovery from all single failures, while also guaranteeing eventual recovery from more major failures. This motivated the current work. The problem of containing the effect of minor failures is becoming important not only due to the improved reliability of system components, but also due to the dramatic growth of network sizes. Such limited transient faults are likely to give rise to states that are almost legal, rather than states that are arbitrary. Hence, it is desirable that the system should recover quickly from such limited transient faults. Moreover, to ensure that non-faulty processes largely remain unaffected by such minor level failures, only a small part of the network around the faulty components should be allowed to make observable state changes.

The tight containment of the effect of single failures depends on the context: *containment in time* implies that all observable variables of the system recover to their legal values in $O(1)$ time, whereas *containment in space* means that the processes at $O(1)$ distance from the faulty process make observable changes. For optimal performance, both of these properties should hold.

An important issue in system design is the mean time between failures, commonly termed the MTBF. Once a fault-containing system recovers from a single failure in constant time, how much time will elapse before the system becomes ready to recover from the next single failure with the same efficiency is an important metric, and it is called the *fault gap* [17]. If the next failure hits the system sooner than this period, then the guarantee of $O(1)$ time recovery falls apart. Thus, the fault gap determines the availability of the fault-free system, and a low fault gap reflects better availability. The dramatic growth of network size increases the probability of failures, and most solutions to fault containment that we know of achieve a fault

* Corresponding author. Tel.: +1 319 335 0738.
E-mail addresses: adasgupt@cs.uiowa.edu (A. Dasgupta), ghosh@cs.uiowa.edu (S. Ghosh), xinxiao@cs.uiowa.edu (X. Xiao).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.02.039

gap of $O(n)$ or worse. As a result, when two single faults occur relatively quickly, the system fails to provide the guarantee of efficient recovery, and in fact the second single failure may sometimes require $O(n)$ (or higher) time for recovery. This seriously undermines the availability of the fault-free system. So a scale-free fault gap is an important design goal.

1.1. Contributions

We show how fault containment can be added to *weakly stabilizing* distributed systems. We present solutions using a randomized scheduler, and illustrate techniques to bias the random schedules so that the system recovers from all single faults in a time independent of the size $n = |V|$ of the system, and the effect of the failure is contained within $O(1)$ distance from the faulty node with high probability (this probability can be controlled by a user-defined tuning parameter). Using this technique, we solve two problems: one is the *persistent-bit problem* on an arbitrary connected topology, and the other is a *leader election problem* on a line topology. Our solutions exhibit a low *fault gap*, and guarantee *high system availability*; once the output variables of the system recover to a legal configuration, the system is immediately ready to handle the next single fault with the same degree of efficiency [4,5]. Our solutions guarantee that the fault gap depends only on the degree of the nodes, and is independent of the size of the network. This significantly increases the *availability* of the fault-free system.

For the *persistent-bit problem*, our solution contains a single fault in $O(\Delta^3)$ expected number of steps, where Δ is the highest degree of a node in the network. For the *leader election problem* on a line topology, our algorithm confines the effect of any single fault to the constant-distance neighborhood of the faulty process. We show that the contamination number (maximum number of processes that change the primary part of their local states during recovery from any single-fault configuration) is at most 4 with high probability. Also, the expected recovery time from a single fault is independent of the network size.

1.2. Related work

Kutten and Peleg [7] introduced a protocol for fault mending that corrects a system from minor failures, but provides no guarantee for stabilization. Subsequently, for specific problems, fault containment has been added to self-stabilization. In [16], Ghosh and Gupta solved the leader election problem on an oriented ring, and presented a simple self-stabilizing leader election protocol that recovers in $O(1)$ time from a 1-fault configuration. Only the faulty node and its two neighbors change their states during convergence to a stable state. In [18], Ghosh et al. presented a fault-containing self-stabilizing protocol for constructing a spanning tree of an arbitrary network. In [13], the same authors demonstrated a fault-containing self-stabilizing protocol for BFS – Breadth First Search tree construction in an arbitrary network. Ghosh et al. [17,8] and Gupta [19] demonstrated a general technique of adding fault containment to self-stabilization, and analyzed the cost of it. Dolev and Herman introduced superstabilizing protocols [12] that, in addition to being stabilizing, guarantee that, during convergence from configurations arising from legal states by small-scale topology changes, certain passage predicates are satisfied. Herman's self-stabilizing protocol [15] for mutual exclusion investigates the possibility of superstabilizing protocols for mutual exclusion in a ring of processors, where a local fault consists of any transient fault at a single processor, and the passage predicate for mutual exclusion specifies that there be at most one token in the ring, with the single exception of a spurious token co-located with the transient fault. Kutten and Patt-Shamir [3] proposed an asynchronous stabilizing algorithm for the persistent-bit problem. Their solution leads to recovery in $O(k)$ time from any k-faulty configuration. A similar protocol for the problem of token passing was addressed by Beauquier et al. [9], where up to k faults hit nodes by corrupting their state undetectably. Beauquier et al. also investigated k-strong self-stabilizing systems [10], which satisfy the properties of strong confinement and of k-linear time adaptivity. Strong confinement means that a non-faulty processor has the same behavior with or without the presence of faults elsewhere in the system. k-linear time adaptivity means that after k or fewer faults hitting the system in a correct state, recovery takes a number of rounds linear in k. Other approaches for fault containment can be found in [1,2].

1.3. Organization of the paper

This paper has five sections. Section 2 describes the models and notation. Section 3 presents the algorithms and the results for the persistent-bit protocol. Section 4 describes the algorithms and the results for the leader election problem. Finally, Section 5 contains some concluding remarks.

2. Models and notation

Let $G = (V, E)$ denote the topology of a distributed system, where V denotes the set of nodes $\{0, 1, \ldots, n-1\}$ representing processes, and $E \subseteq (V \times V)$ represents the set of edges. Each edge $(i, j) \in E$ represents a bidirectional link between processes i and j. We use the notation $N(i)$ to represent the neighbors of i: thus $(i, j) \in E \Leftrightarrow j \in N(i)$. Processes communicate with their immediate neighbors (also called the distance-1 neighbors) using the *shared memory* model. The execution of the protocols is organized in steps. In each step, a process i executes one or more guarded actions $g \rightarrow A$, where g is a predicate involving the variables of i and $N(i)$, and A is an action that updates one or more variables of i. The global

state or configuration S of the system is a tuple consisting of the local states of all the processes. Unless otherwise stated, a central scheduler (also called a *central demon*) serializes all guarded actions. We assume a randomized scheduler, where the central demon *randomly chooses* an action with an enabled guard with uniform probability. The scheduler is weakly fair. A *computation* of the system is a finite or infinite sequence of global states that satisfies two properties: (a) if S and S' are two consecutive states in the sequence, then there exists a process i such that i has an enabled guard in S and execution of the corresponding action results in the state S', and (b) if the sequence is finite, then, in the last state of the sequence, no process has an enabled guard.

Definition 1 (*Weak Stabilization*)**.** *Weak stabilization* [14] guarantees that, starting from an arbitrary configuration, there exists *at least one computation* that leads the system to a *legal* configuration.

A stabilizing system converges to a legal configuration LC that is traditionally defined in terms of the observable or *primary* variables. However, in most cases, fault containment requires the use of auxiliary or *secondary* variables too. We define the local state of each process i as an ordered pair $\langle p_i, s_i\rangle$, where p_i denotes the primary variables, and s_i denotes the secondary variables. Correspondingly, we write the global state as an ordered pair $\langle p, s\rangle$, where p is the collection of all primary variables and s is the collection of all secondary variables. For any global state $\langle p, s\rangle$, $\langle p, s\rangle \in LC \Rightarrow p \in LC_p$ and $s \in LC_s$. The following are some important definitions relevant to the fault-containment problem [18].

Definition 2 (*Stabilization Time*)**.** The *stabilization time* is the maximum time needed to establish LC from an arbitrary initial configuration.

Though $LC \Rightarrow p \in LC_p$ and $s \in LC_s$, in most cases, it is adequate to establish LC_p only, since the application is not affected by the secondary variables.

Definition 3 (*Fault Containment in Time*)**.** *Fault containment in time* means that, starting from any single-fault configuration, LC_p is restored in $O(1)$ time.

This definition may appear strict in some cases. So we define the notion of *weak fault containment*.

Definition 4 (*Weak Fault Containment*)**.** A system is said to be weakly fault containing if the containment time of the system is independent of n, i.e., the size of the network.

Note that, for weak fault containment, LC_p does not necessarily have to be restored in $O(1)$ time.

Definition 5 (*Fault Containment in Space*)**.** *Fault containment in space* means that, starting from a single-fault configuration, the primary variables of the processes at $O(1)$ distance from the faulty process make observable changes.

One can also define a weaker version of fault containment in space, where the distance up to which the primary variables are affected is independent of n.

Ideally, attempts must be made to restrict the effect of the fault within the immediate neighborhood of the original faulty node, which is the notion of tight containment. This way, the rest of the system can still perform its normal operation. The fault should not contaminate a large portion of the network.

Definition 6 (*Contamination Number*)**.** The *contamination number* is the maximum number of processes that change the primary part of their local states during recovery from any single-fault configuration.

Ideally, the contamination number should be as small as possible.

Definition 7 (*Fault gap*)**.** Starting from any single-fault configuration, the *fault gap* is the maximum time required to reach a state in LC.

The fault gap determines the availability of the fault-free system, and a low fault gap reflects better availability. If the next failure hits the system sooner than the time duration of the fault gap, then the guarantee of constant time recovery does not hold anymore.

In a probabilistic setting, the definitions of stabilization time and contamination number will be weakened to reflect their expected values only.

3. Persistent-bit protocol

For the sake of exposition, we consider the case of *persistent-bit* protocol, in which a set of processes maintains the value of a replicated bit $v \in \{0, 1\}$ across a connected network. There are two distinct legal configurations: all 0s and all 1s. The protocol in Algorithm 1 is weakly stabilizing.

Algorithm 1 Persistent-bit protocol: Program for process i

A1. **do** $\exists j \in N(i) : v(j) \neq v(i) \rightarrow v(i) := v(j)$ **od**

Lemma 1. *The persistent-bit protocol is not fault containing with a randomized demon.*

Proof. Consider a linear array of processes numbered $0, 1, \ldots, n-1$ from left to right, and assume that initially $\forall i : v(i) = 1$ holds. Let a failure of process 0 change $v(0)$ to 0. With this as the starting state, the computation can be reduced to a run of *gambler's ruin*[1]: whenever a process with $v = 0$ executes an action, the boundary between the dissimilar values of v shifts to the left, and whenever a process with $v = 1$ executes an action, the boundary moves to the right. The game is over when the system reaches LC, and per [11] the expected number of moves needed is $(1 \times n - 1)$, i.e., $O(n)$. Thus, the protocol is *not* fault containing. □

In view of Lemma 1, we present a fault-containing version of the persistent-bit protocol.

3.1. Fault-containing persistent-bit protocol

To make the protocol fault containing, we add to each process i a secondary variable $x(i) \in \mathbb{Z}^*$ (the set of non-negative integers). In a way, $x(i)$ will reflect the priority of process i in executing an action to update $v(i)$. Process i will update $v(i)$, when the following three conditions hold.

1. The randomized scheduler chooses i,
2. $\exists j \in N(i) : v(j) \neq v(i)$, and
3. $\forall j \in N(i) : x(i) \geq x(j)$.

After updating $v(i)$, process i will increase $x(i)$ to $\max\{x(j) : j \in N(i)\} + m$, where m is a constant positive integer. When only the first two conditions hold, but not the third, then process i will increment the value of $x(i)$ by 1, and leave $v(i)$ unchanged. Algorithm 2 shows the modified protocol.

Algorithm 2 Probabilistic fault-containing algorithm: Program for process i

A1. $\exists j \in N(i) : v(j) \neq v(i) \wedge \forall k \in N(i) : x(i) \geq x(k) \rightarrow v(i) := v(j); x(i) := \max\{x(k) : k \in N(i)\} + m$
A2. $\exists j \in N(i) : v(j) \neq v(i) \wedge \exists\, k \in N(i) : x(i) < x(k) \rightarrow x(i) := x(i) + 1$
A3. $\forall j \in N(i) : v(j) \neq v(i) \rightarrow v(i) := v(j)$

Observe that once a process i updates $v(i)$, it becomes difficult for its neighbors to change their v-values, since their x-values will lag behind that of i. The larger is the value of m, the greater is the difficulty. A neighbor j of i will be able to update $v(j)$ only if it is chosen by the random scheduler m times, without choosing i even once. On the other hand, it becomes easier for i to update $v(i)$ again in the near future.

Failures can not only corrupt v, but also corrupt x. Assume that $LC \equiv \forall j : v(j) = 1$, and that a single failure at process i changes $v(i)$ to 0 and $x(i)$ to some unknown value. If $\forall j \in N(i) : x(i) > x(j)$, then process i is likely to change its $v(i)$ soon again, before its neighbors get a chance to do so. As a result, the fault is contained in a small number of steps, and the contamination number is 1. However, a smart adversary injecting the failure at process i is likely to set $x(i)$ to the smallest value (i.e., 0). This makes the neighbors of process i better candidates for changing their v, before process i executes a move to complete the recovery. However, it also raises the x-values of these neighbors of i above those of *their* neighbors. In order that the fault percolates to a node at distance 2 from the faulty process i, such a distance-2 node has to be chosen by the scheduler at least m times, without choosing its neighboring distance-1 node even once. With a large value of m, the probability of such an event is very low. This explains the mechanism of containment. In the meantime, the condition $\forall j \in N(i) : v(j) = 1$ is likely to hold several times. If on one such occasion the faulty process is chosen by the random scheduler (action $A3$; note that its guard does not depend on x), then $v(i)$ will change to 1, and the recovery will be complete.

3.2. Results

We begin with an analysis of the spatial containment. Assume that all nodes have a degree Δ. Then the following theorem holds.

Theorem 1. *As $m \rightarrow \infty$, the effect of a single failure is restricted to only the immediate neighbors of the faulty process.*

Proof. Suppose that the faulty process has n_1 distance-1 neighbors and n_2 distance-2 neighbors. The probability that a distance-2 neighbor is contaminated is largest, when only one distance-1 process is contaminated, and only one neighbor of that contaminated distance-1 process (which is a distance-2 neighbor of the faulty process) is contaminated. The probability of one distance-1 neighbor being contaminated is $\frac{n_1}{n_1+1}$. To contaminate a distance-2 neighbor, the scheduler must select the specific process m times. So the probability of one distance-2 neighbor being contaminated is $\frac{1}{(n_1+n_2+1)^m}$. Therefore, after a node becomes faulty, the probability that some distance-2 neighbor of the faulty process becomes contaminated is $\frac{n_1}{n_1+1} \times n_2 \times \frac{1}{(n_1+n_2+1)^m}$. By choosing a large value of m, this probability can be made as small as possible. □

[1] The original study is by Coolidge [11] in 1909, where he showed that if two gamblers start with capitals of x and $N - x$, and each fair coin toss transfers a dollar from one to the other depending on the outcome of the toss, then the expected number of steps to finish the game is $x \cdot (N - x)$.

Theorem 2. *If $\Delta << m$, then the expected number of steps needed to contain a single fault is $O(\Delta^3)$.*

Proof. Since m is very large, per Theorem 1, the effect of the fault is unlikely to propagate to the distance-2 neighbors of the faulty process. It is therefore sufficient to consider the case when the fault propagates to the immediate neighbors of faulty process.

Since there are at most Δ neighbors of the faulty process, the system can have at most $\Delta + 1$ faults. Let $p_{i,j}$ denote the probability that the number of the faults changes from i to j due to the action taken by some process. Below, we list such probabilities for various possible values of i and j:

$$p_{1,0} = \frac{1}{\Delta + 1} \tag{1}$$

$$p_{2,1} = \frac{1}{2\Delta} \tag{2}$$

$$p_{2,2} = \frac{\Delta}{2\Delta} \tag{3}$$

$$p_{1,2} = \frac{1}{\Delta + 1} \tag{4}$$

and for $3 \leq i \leq \Delta - 1$

$$p_{i,i+1} = \frac{i}{(\Delta + 1) + i(\Delta - 1)} \tag{5}$$

$$p_{i+1,i+1} = \frac{1 + i(\Delta - 1)}{(\Delta + 1) + i(\Delta - 1)} \tag{6}$$

$$p_{i+1,i+2} = \frac{\Delta - 1}{(\Delta + 1) + i(\Delta - 1)} \tag{7}$$

and

$$p_{\Delta+1,\Delta} = \frac{\Delta - 1}{(\Delta + 1) + \Delta(\Delta - 1)} \tag{8}$$

$$p_{\Delta+1,\Delta+1} = \frac{1 + \Delta(\Delta - 1)}{(\Delta + 1) + \Delta(\Delta - 1)}. \tag{9}$$

We use $P[X]$ to denote the probability that the system recovers using X moves. The expected number of moves needed is

$$\begin{aligned} E &= 1 \times P[1] + 2 \times P[2] + 3 \times P[3] + \cdots \\ &= \sum_{x=1}^{\infty} XP[X]. \end{aligned}$$

Starting from a 1-fault configuration, during a recovery phase, the number of faulty processes may initially grow, but must eventually shrink. Accordingly, we first calculate $P[X]$ using (1), (2), and (4), as follows:

$$P[1] = p_{1,0} = \frac{1}{\Delta + 1} \tag{10}$$

$$P[2] = 0 \tag{11}$$

$$P[3] = p_{1,2}p_{2,1}p_{1,0} = \frac{1}{\Delta + 1}\frac{1}{2\Delta}\frac{1}{\Delta + 1} = \frac{1}{2\Delta(\Delta + 1)^2} \tag{12}$$

$$P[4] = p_{1,2}p_{2,2}p_{2,1}p_{1,0} = \frac{1}{\Delta + 1}\frac{\Delta}{2\Delta}\frac{1}{2\Delta}\frac{1}{\Delta + 1}, \tag{13}$$

and, for $n \geq 1$, we get the following recursive function of $P[2n + 2]$ using $P[2n + 1]$:

$$P[2n + 2] = 2\sum_{j=2}^{m} p_{j,j}P[2n + 1]. \tag{14}$$

(**Note:** m and n used here are variable parameters only, and should not be confused with the scheduler bias or the size of the system.) If $n + 1 < \Delta + 1$, then $m = n$. If $n + 1 > \Delta + 1$, then $m = \Delta + 1$. This is because, if $n + 1 < \Delta + 1$, using $2n + 2$ steps, the system can reach at most the $n + 1$-th state. So the repeated moves can occur in any state within the n-th state. But if $n + 1 > \Delta + 1$, the system can move through all the states, so the repeated moves can occur anywhere within the $\Delta + 1$ states.

We can also write $P[2n+3]$ using $P[2n+1]$ as

$$P[2n+3] = 2\sum_{j=2}^{m}\sum_{i=2}^{m} p_{j,j}p_{i,i}P[2n+1] + 2\sum_{k=2}^{m'} p_{i,i+1}p_{i+1,i}P[2n+1]. \tag{15}$$

The values of m and m' are the same as in (14). The system can have either two additional moves that do not change the current state, or the system can first reach a state in one step and come back in the next step. We now substitute for $P[1]$, $P[2]$, $P[3]$, *and*$P[4]$ using (10)–(15) in (10), and let $p_{max} = \max\{p_{i,j}, \forall i, \forall j\}$:

$$\begin{aligned}
E &= 1\times P[1] + 2\times P[2] + 3\times P[3] + 4\times P[4] \\
&\quad + \sum_{n=2}^{\infty}\{(2n+1)P[2n+1] + (2n+2)P[2n+2] + (2n+3)P[2n+3]\} \\
&= 1\times\frac{1}{\Delta+1} + 2\times 0 + 3\times\frac{1}{2\Delta(\Delta+1)^2} + 4\times\frac{1}{4\Delta(\Delta+1)^2} \\
&\quad + \sum_{n=2}^{\infty}\left\{(2n+1) + 2(2n+2)\sum_{j=2}^{m}\sum_{i=2}^{m} p_{i,i}p_{j,j} + (2n+3)\left(2\sum_{i=2}^{m'} p_{i,i+1}p_{i+1,i} + 2\sum_{j=2}^{m''}\sum_{i=2}^{m''} p_{i,i}p_{j,j}\right)\right\}\times P[2n+1] \\
&< O\left(\frac{1}{\Delta^2}\right) + \sum_{n=2}^{\infty}\{(2n+1)p_{max}^{2n+1} + 2(2n+2)\Delta^2 p_{max}^{2n+2} + (2n+3)(2\Delta+2\Delta^2)p_{max}^{2n+3}\} \\
&= \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1} + 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2} + (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3} + O\left(\frac{1}{\Delta^2}\right).
\end{aligned}$$

Let $T_1 = \sum_{n=2}^{\infty}(2n+1)p_{max}^{2n+1}$, $T_2 = 2\Delta^2\sum_{n=2}^{\infty}(2n+2)p_{max}^{2n+2}$, $T_3 = (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3}$. As the order of T_3 is the same as that of T_2 and larger than that of T_1, we just need to calculate T_3.

$$\begin{aligned}
T_3 &= (2\Delta+2\Delta^2)\sum_{n=2}^{\infty}(2n+3)p_{max}^{2n+3} \\
&= (2\Delta+2\Delta^2)\left(2p_{max}^3\sum_{n=2}^{\infty}np_{max}^{2n} + 3p_{max}\sum_{n=2}^{\infty}p_{max}^{2n}\right) \\
&= (2\Delta+2\Delta^2)\left[2p_{max}^3\left(\frac{2p_{max}^4}{1-p_{max}^2} + p_{max}^6\right) + 3p_{max}^3\frac{1}{1-p_{max}^2}\right] \\
&= (2\Delta+2\Delta^2)\frac{4p_{max}^7 + 2p_{max}^3p_{max}^6(1-p_{max}^2) + 3p_{max}^3}{1-p_{max}^2}.
\end{aligned}$$

Let $p_{max} = \frac{a}{\Delta+a}$, $a = 1+\Delta(\Delta-1)$:

$$\begin{aligned}
T_3 &= (2\Delta+2\Delta^2)\frac{4a^7(\Delta+a)^2 + 2a^9 + 3a^3(\Delta+a)^6}{(\Delta^2+2\Delta a)(\Delta+a^7)} \\
&= O(\Delta^2)\times\frac{O(a^9)}{O(\Delta^3)O(a^7)} \\
&= \frac{O(a^2)}{O(\Delta)} \\
&= O(\Delta^3).
\end{aligned}$$

So we get

$$E = O\left(\frac{1}{\Delta^2}\right) + O(\Delta^3) = O(\Delta^3). \quad \square \tag{16}$$

When the graph is dense, i.e., $\Delta = O(n)$, the containment time is not independent of the size of the network anymore. However, the spatial containment property still holds. The more dense the graph is, the smaller is the contamination number. As $m \to \infty$, contamination number tends to 1. Below, we separately analyze the extreme case of a dense topology: a completely connected graph.

Theorem 3. *For a completely connected graph, the contamination number approaches* 1 *as* $m \to \infty$.

Proof. Let i be the faulty process. At least one neighbor j of the faulty process i is likely to update $v(j)$, and raise $x(j)$ at least m steps above the x-values of the rest. The contamination number can increase beyond 1 only if a second neighbor k updates $v(k)$ (instead of the system recovering to its legal configuration). This requires the scheduler to choose the neighbor k at least m times, without choosing i or j even once.

Let $P\{n_0, n_1, n_2, \ldots, n_i, n_{i+1}, \ldots, n_{\Delta-1}\}$ denote the probability that, $\forall i, 1 \leq i \leq \Delta$, node i is chosen n_i times. So the probability of node k being chosen m times before j being chosen even once is

$$P\{n_0, n_1, n_2, \ldots, n_{j-1}, n_j = 0, n_{j+1}, \ldots, n_k = m, \ldots, n_{\Delta-1}\} \quad \forall i, 0 \leq i \leq \Delta - 1 \wedge i \neq j, n_i \leq m. \tag{17}$$

With increasing n_i, $1 \leq i \leq \Delta \wedge i \neq j$, (17) is decreasing (see Lemma 2 for the proof). So the above probability is maximum when node k is consecutively chosen m times, and other nodes are never chosen. The maximum probability is

$$P\{0, 0, \ldots, 0, m, 0, \ldots, 0, 0\} = \frac{1}{\Delta^m}. \tag{18}$$

Since n_i, $0 \leq i \leq \Delta - 1 \wedge i \neq j$ can be any value between 0 and m, and there are in total $m^{\Delta-2}$ possible situations, we apply the maximum estimate to each such case. As a result, the probability of the system having two contaminated processes is no larger than $\frac{m^{\Delta-2}}{\Delta^m}$, which approaches 0 as m approaches ∞. □

Lemma 2. *For a completely connected graph, where i is a faulty node, the probability that node k is chosen m times before node j is chosen once is maximum when node k is consecutively chosen m times and other nodes are never chosen.*

Proof. The probability that node k is chosen m times before node j is chosen once is

$$\begin{aligned} P\{n_k = m, n_j = 0\} &= \sum_{i=1 \wedge i \neq j \wedge i \neq k}^{\Delta} \sum_{n_i=0}^{m} P\{n_1, n_2, \ldots, n_j = 0, \ldots, n_k = m, \ldots, n_\Delta\} \\ &= \binom{\sum_{i=1}^{\Delta} n_i}{n_1} \binom{\sum_{i=2}^{\Delta} n_i}{n_2} \cdots \binom{\sum_{i=j}^{\Delta} n_i}{0} \cdots \binom{\sum_{i=k}^{\Delta} n_i}{m} \cdots \binom{\sum_{i=\Delta}^{\Delta} n_i}{n_\Delta} \left(\frac{1}{\Delta}\right)^{\sum_{i=1}^{\Delta} n_i} \\ &= l \times \left(\frac{1}{\Delta}\right)^q, \end{aligned}$$

where $l = \binom{\sum_{i=1}^{\Delta} n_i}{n_1} \cdots \binom{\sum_{i=\Delta}^{\Delta} n_i}{n_\Delta}$, $q = \sum_{i=1}^{\Delta} n_i$. If we increase any n_i, $1 \leq i \leq \Delta \wedge i \neq j \wedge i \neq k$ to $n_i + 1$, we can see this selection process as follows: first do the selection in the same way as before we increase n_i, then choose any one of the eligible processes. There are in all $\Delta - 1$ processes that can be chosen for the last step, and the probability of choosing any one of them is $\frac{1}{\Delta}$, so the probability will be $l \times (\Delta - 1) \times \left(\frac{1}{\Delta}\right)^{q+1}$, and this is smaller than $l \times \left(\frac{1}{\Delta}\right)^q$. So the probability that node k is chosen m times before choosing node j chosen once will decrease as n_i increases. □

The above mechanism reveals that a high clustering coefficient limits the probability of contamination to only a small fraction of the distance-1 neighbors. The completely connected graph exhibits an extreme form of this property.

3.3. Computing the availability

An interesting aspect of the proposed algorithm is that $LC_s = true$; thus there is no overhead for stabilizing the secondary variables. So LC holds as soon as LC_p holds. This leads to the following theorem.

Theorem 4. *For single failures, the fault gap equals the containment time.*

As a consequence of this, within an expected time of $O(\Delta^3)$ after each single failure, the system is ready to withstand the next single failure with the same efficiency. Furthermore, since only the distance-1 neighbors are contaminated with high probability, the proposed algorithm enables the system to recover from all concurrent failures of nodes that are distance 3 or more apart with the same efficiency. This significantly increases the availability of the system compared to existing solutions that we know of.

A drawback of the proposed solution is that the values of the x-variables grow in an unbounded manner, and this affects the implementability of the protocol. One way of overcoming this problem is transforming the solution into one that relies on bounded variables only [4].

Theorem 5. *Algorithm 2 is weakly stabilizing.*

3.4. Experimental results

We ran simulation experiments on random graph topologies $G(n, p)$ with $n = 1000$ nodes and edge probability p set to 0.5 for various values of m ranging from 2 to 32. The initial values of x were randomly chosen. The fraction of cases where

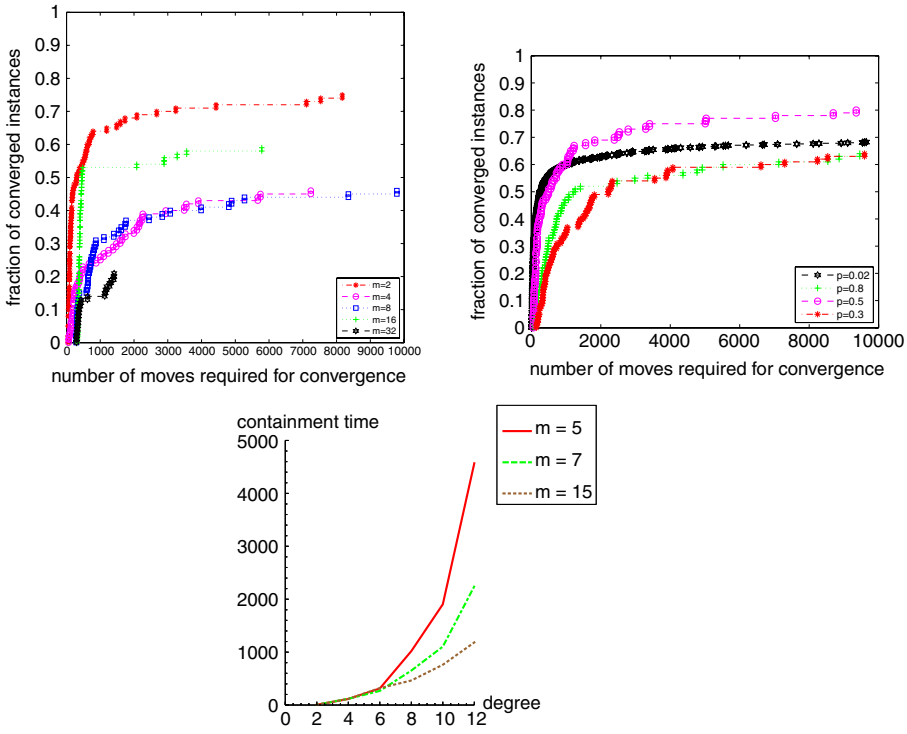

Fig. 1. Simulation results. (a) The number of moves required for convergence (X-axis) against the fraction of converged instances (Y-axis) with varying m. Because of the probabilistic nature of our algorithm, there were instances when the system of nodes did not converge. (b) The number of moves required for convergence (X-axis) against the fraction of converged instances (Y-axis) for different values of p. (c) The relationship between stabilization time and node degree for various values of m.

the fault was successfully contained was plotted. Because of the probabilistic nature of our algorithm, there were instances when the system of nodes did not converge. We set an upper bound (10,000 moves) for the number of moves required for the system to converge. If convergence had not occurred within this bound, then those instances were treated as failed cases (i.e., non-convergence). Fig. 1(a) shows the number of moves required for convergence (X-axis) against the fraction of converged instances (Y-axis) for various values of m.

We expect the value of m to define the effectiveness of a *barrier* around the faulty node, preventing the propagation of the effect of the failure. For $m = 2$ and $m = 32$, the two extreme m-values we had chosen in our experiments, the system endorses the expected behavior. Note that when m is large ($m = 32$), the fraction of converged instances decreases; however, these instances of convergence take fewer number of moves, since the effect of the failure does not propagate beyond the immediate neighbors. On the other hand, if the effect of the fault is able to cross the 'barrier' set by m, then recovery takes much longer (for $m = 32$, just above 20% of cases converged within 10 000 moves). But for mid-range values of m, an anomaly was observed. For example, the graph for $m = 16$ increases above those for $m = 8$ and $m = 4$. It appears that there is a threshold value of m beyond which the system may be in a "stuttering mode", since neither does the fault propagate beyond the distance-1 neighbors, nor is the recovery complete. In our case, this phenomenon was observed between $m = 4$ and $m = 8$. Therefore, the fraction of converged instances for those values lags behind.

In the second set of experiments, we set $m = 2$. Again the initial values of x were randomly set. This time we varied p, which denotes the edge probability for the random graph, and plotted the number of moves required for convergence (X-axis) against the fraction of converged instances (Y-axis). The corresponding distribution is shown in Fig. 1(b). It captures how node degrees can influence the convergence of the system. For a Δ-ary tree, this is equivalent to varying the value of Δ, while keeping m fixed. As we used random graphs for our experiments, we could not directly vary Δ. Rather, we chose different values of p to control the sparseness/density of the graphs. As expected, for dense graphs (e.g., $p = 0.8$), the fraction of converged instances was low.

For the third set of experiments (Fig. 1(c)), the topology was a multi-dimensional torus with each node having a degree of Δ, and in each dimension there were 9 nodes (so $n = 9^{\Delta}$). As before, the initial values of x were randomly chosen. Fig. 1(c) shows the containment time for various values of Δ. The results reveal that the *stabilization time* (the same as the containment time) increases for smaller values of m, since the fault contaminates a small number of nodes beyond the distance-1 neighbors. (The existence of a 'knee' in these curves corresponds to the event when the effect of the fault spills over the distance-1 neighbors.) When Δ increases (but the fault is successfully contained), for a long time, the system may be in a "stuttering mode", since neither does the fault propagate beyond the distance-1 neighbors, nor is the recovery complete. When the value of m is small, the 'knee' appears at a lower degree. This is expected, because both larger m and higher values

of Δ adversely affect fault propagation as well as convergence. An additional experiment conducted on a (30×30) grid revealed that, when $m = 5$, the fault is successfully contained within a neighborhood of size 3 in 96% of the cases, but for $m = 10$ this number exceeds 99%.

4. Algorithm for leader election

We now extend our technique to design a probabilistic solution to fault containment for a different problem, that of leader election. Let $A = (V, E)$ denote an array of a distributed system, where $V = \{0, 1, \ldots, n-1\}$ represents the set of processes, and each edge $(i, j) \in E$ represents a link between processes i and j. Each process i has a parent variable $P(i)$, where $P(i) \in N(i) \cup \perp$, and a secondary variable $x(i) \in \mathbb{Z}^+$. The purpose of the secondary variable is to contain the fault. $\forall j \in N(i) : P(j) = i \wedge P(i) = \perp$ implies that i is the leader. In a legal configuration, there can only be a single leader in the system. Processes communicate with their immediate neighbors (also called the distance-1 neighbors) using the shared memory model.

Our starting point is the weakly stabilizing leader election algorithm on a tree network presented by Devismes et al. [6]. An array is a special case of a tree where each node except the end nodes has a degree of 2. Therefore we disregard the notation Δ from [6], and just consider the two neighbors (or the neighbor if it is an end node) of a particular process. The original algorithm has three rules. In Algorithm 3, we reproduce the algorithm of Devismes et al. adapted to a line topology.

Algorithm 3 Weakly stabilizing leader election algorithm of Devismes et al.: Program for process i

- Variable: $P(i) \in N(i) \cup \{\perp\}$.
- Macro: $C(i) = \{q \in N(i) | P(q) = i\}$
- Predicates: $Leader(i) \equiv (P(i) = \perp) \wedge (\forall j \in N(i) : P(j) = i)$
- Actions:

R1. $(P(i) \neq \perp) \wedge (|C(i)| = |N(i)|) \longrightarrow P(i) \leftarrow \perp$
R2. $(P(i) \neq \perp) \wedge (\exists k \in N(i) \setminus \{C(i) \cup P(i)\}) \longrightarrow P(i) \leftarrow k$
R3. $(P(i) = \perp) \wedge (|C(i)| < |N(i)|) \longrightarrow P(i) \leftarrow (N(i) \setminus C(i))$

We directly use R1 and R3 from [6] in our new algorithm (R1 and R2 in Algorithm 3), but, for the fault-containment part, we need to add new rules, consistent with the basic approach used in the persistent-bit protocol and also described in [4]. Therefore, R2 of [6] is extended to multiple rules in Algorithm 4 (R3, R4, and R5). To make the protocol fault containing, we add to each process i a secondary variable $x(i) \in \mathbb{Z}^+$. In a way, $x(i)$ will reflect the priority of process i in executing an action to update $P(i)$. Our fault-containing algorithm is shown in Algorithm 4. Process i will update $P(i)$ and increase its $x(i)$-value with respect to its neighbors when the following conditions hold.

1. The randomized scheduler chooses i,
2. $\{(\exists j \in N(i) : P(i) = j)\}$,
3. $\{(\exists k \in N(i) : P(k) = l \neq i)\}$,
4. $\{x(i) \geq x(k)\}$.

After updating $P(i)$, process i will increase its $x(i)$-value accordingly: $\{x(i) \leftarrow \max_{q \in N(i)} x(q) + m\}$, $m \in \mathbb{Z}^+$. For simplicity of exposition, we use unbounded m.

If the first three conditions hold, but not the fourth one, process i increases its x_i-value by 1, and leaves $P(i)$ unchanged. Process i also increases its $x(i)$-value by 1, and leaves $P(i)$ unchanged when the following conditions hold.

1. The randomized scheduler chooses i,
2. $\{(\exists j \in N(i) : P(i) = j)\}$,
3. $\{(\exists k \in N(i) : P(k) = \perp)\}$,
4. $\{x(i) < x(k)\}$.

Observe that, once a process i updates $x(i)$, it becomes difficult for its neighbors to change their P-values, since their x-values will lag behind that of i. The larger the value of m is, the greater is the difficulty. A neighbor j of i will be able to update $P(j)$ if it is chosen by the random scheduler m times, without choosing i even once (except case R5, where the update takes place immediately when recovery is in sight within a single future move). On the other hand, it becomes easier for i to update $P(i)$ again in the near future. With a large value of m, the probability of j changing its parent pointer compared to i is very low. This explains the mechanism of containment.

In Algorithm 4:

1. R1 describes the situation when a process i has a parent but all its neighbor(s) consider(s) i as its(their) parent. So i sets its parent pointer to null and start considering itself as the leader.
2. R2 describes the situation when a process i has no parent and one of its neighbors q does not satisfy the condition $P(q) = i$. Note that for a single-fault scenario it cannot be the case that both of i's neighbors do not satisfy the same condition. This means that i is not unanimously selected as the leader by its neighbors. As a consequence, i stops considering itself as a leader by setting its parent pointer to q, i.e., $P(i) = q$.

Algorithm 4 *Probabilistic fault-containing leader election algorithm: Program for process i*

- Variable: $P(i) \in N(i) \cup \{\perp\}$.
- Macro: $C(i) = \{q \in N(i) | P(q) = i\}$
- Predicates: $Leader(i) \equiv (P(i) = \perp) \wedge (\forall j \in N(i) : P(j) = i)$
- Actions:

R1. $(P(i) \neq \perp) \wedge (|C(i)| = |N(i)|) \longrightarrow P(i) \leftarrow \perp$
R2. $(P(i) = \perp) \wedge (|C(i)| < |N(i)|) \longrightarrow P(i) \leftarrow (N(i) \setminus C(i))$
R3. (a) $(\exists j \in N(i) : P(i) = j) \wedge (P(j) \neq \perp) \wedge (\exists k \in N(i) : P(k) = l \neq i \text{ or } \perp) \wedge (x(i) \geq x(k)) \longrightarrow (P(i) \leftarrow k) \wedge \left(x(i) \leftarrow \max_{q \in N(i)} x(q) + m\right), m \in \mathbb{N}$
(b) $(\exists j \in N(i) : P(i) = j) \wedge (P(j) \neq \perp) \wedge (\exists k \in N(i) : P(k) = l \neq i) \wedge (x(i) < x(k)) \longrightarrow x(i) \leftarrow x(i) + 1$
R4. (a) $(\exists j \in N(i) : P(i) = j) \wedge (\exists k \in N(i) : P(k) = \perp) \wedge (x(i) \geq x(k)) \longrightarrow P(i) \leftarrow k$
(b) $(\exists j \in N(i) : P(i) = j) \wedge (\exists k \in N(i) : P(k) = \perp) \wedge (x(i) < x(k)) \longrightarrow x(i) \leftarrow x(i) + 1$
R5. $(\exists j \in N(i) : P(i) = j) \wedge (P(j) = \perp) \wedge (\exists k \in N(i) : P(k) = l \neq i \text{ or } \perp) \longrightarrow P(i) \leftarrow k$

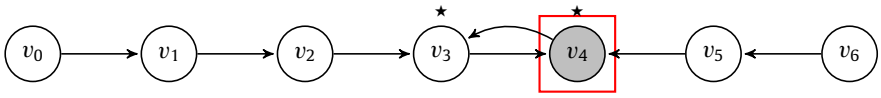

Fig. 2. Fault at leader v_4. Due to the fault, v_3 becomes v_4's parent.

3. R3(a) describes the situation when parent of i is j, and i has a neighbor k whose parent is a node l. Node l is at distance 2 from i. Now, if the x-value of i is no less than that of k, i sets k as its new parent and increases its $x(i)$-value with respect to its neighbors.
4. R3(b) describes the situation when the parent of i is j, j has a parent, and i has a neighbor k whose parent is a node l. Node l is at distance 2 from i. Now, if the x-value of i is smaller than that of k, i does not alter its parent pointer but it just increments its x-value by 1.
5. R4(a) describes the situation when the parent of i is j, and i has a neighbor k whose parent pointer is set to null. Now, if the x-value of i is no less than that of k, i sets k as its new parent.
6. R4(b) describes the situation when the parent of i is j, and i has a neighbor k whose parent is set to null. Now, if the x-value of i is smaller than that of k, i does not alter its parent pointer but it just increments its x-value by 1.
7. The intuition behind R5 is that, if a node finds out that its change of parent will help the system to recover in a single future move, then it makes the move. When the parent of i is j, and j has no parent, and there is a neighbor k of i such that k's parent is a node l, where node l is at distance 2 from i, then even if the x-value of i is smaller than that of k, i sets k as its new parent.

R3, R4, and R5 are actions to contain the fault and expedite the recovery. The effect of the moves will become clear in the next section, when we describe the recovery mechanism in detail.

4.1. Recovery

In this section, we describe the different cases of single-fault configuration and the steps by which the system recovers from single faults. The recovery mechanism shows that the algorithm is fault containing. The fault can occur either at the leader, or at distance 1 from the leader, or at distance 2 from the leader, and so on. It is to be noted that we have to consider cases up to distance 4 from the leader as, beyond distance 4, all cases of recovery are similar to each other for single-fault configuration. For convenience, we consider an array of length n and denote process i on the array as v_i, where $i = 0, \ldots, n-1$. In each figure, the grey node denotes the original leader in the system, the node with a red square is the node hit by the single fault, and the nodes with a $\star$ above are the nodes whose guards are true after the single fault hits the system. A parent–child relation is denoted by a directed arrow;, i.e., if there is an arrow from i to j, this indicates that $P(i) = j$.

4.2. Fault at the leader

We begin with the case when the fault occurs at the leader. In Fig. 2, v_4 is the leader where the fault hits. Let us say that the fault renders v_3 as the parent of v_4; i.e., $P(v_4) = v_3$. In this case, the system recovers trivially in a single step. If the scheduler chooses either of v_3 or v_4, R1 can be applied at v_3 or v_4, and a legal configuration can be reached in a single step. Note that, if v_4 makes the move, the system goes back to the initial legal configuration, whereas if v_3 makes the move, v_3 becomes the new leader of the system after recovery.

4.3. Fault at distance 1 from the leader

(a) The parent pointer of the distance-1 neighbor from the leader becomes null

Let us say that the fault hits v_3 (Fig. 3). If v_3's parent pointer becomes null, again recovery takes place trivially in a single

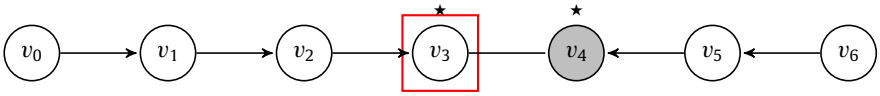

Fig. 3. Fault at distance 1 from the leader. v_3's parent pointer becomes null due to the fault.

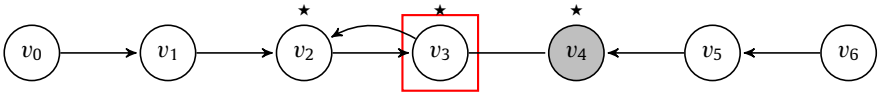

Fig. 4. Fault at distance 1 from the leader. Due to the fault, v_2 becomes v_3's new parent.

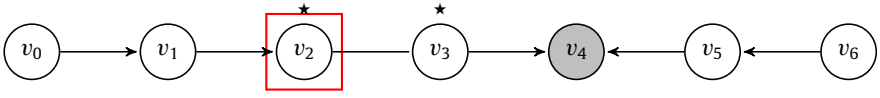

Fig. 5. Fault at distance 2 from the leader. v_2's parent pointer becomes null due to the fault.

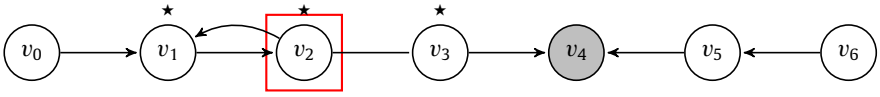

Fig. 6. Fault at distance 2 from the leader. v_1 becomes the new parent of v_2 due to the fault.

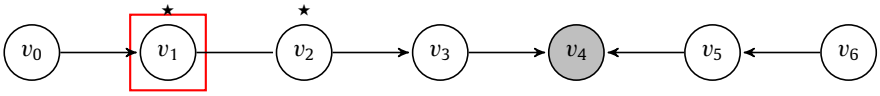

Fig. 7. Fault at distance 3 from the leader. v_1's parent pointer becomes null due to the fault.

step. If the scheduler chooses either of v_3 or v_4, R2 can be applied at v_3 or v_4, and a legal configuration can be reached in a single step. Note that, if v_3 makes the move, the system goes back to the initial legal configuration, whereas if v_4 makes the move, v_3 becomes the new leader of the system after recovery.

(b) A new node becomes the parent of the distance-1 faulty node

Let us consider Fig. 4. If v_2 is the new parent of v_3, i.e., $P(v_3) = v_2$, then, in this case, the guards of v_2, v_3, and v_4 are true. If v_3 is selected first, and if $x(v_3) \geq x(v_4)$, the system trivially recovers in a single step as R4(a) can be applied at v_3. Otherwise, v_3 would not make a parent change. If the scheduler chooses v_4, then the recovery takes place in two steps. First, v_4 applies R2, and after that either v_2 or v_3 makes a move by applying R1. Note that, if v_3 makes the move, the leader shifts one place compared to the original legal configuration, whereas if v_2 makes the move, the leader shifts two places compared to the original legal configuration.

If v_2 makes the first move after being chosen by the scheduler, R1 can be applied at v_2. After that, if the scheduler chooses v_4, recovery is immediate as R2 can be applied at v_4. But if the scheduler chooses v_3 repeatedly and $x(v_3) \geq x(v_4)$ and $x(v_3) \geq x(v_2)$, oscillations can occur in the system for some period of time due to the fact that R4(a) is applicable at v_3. v_2 and v_4 alternately become v_3's parent. But whenever v_2 or v_4 is chosen by the scheduler next, the system recovers by applying R2. When $x(v_3) < x(v_2)$ or $x(v_3) < x(v_4)$, recovery is complete when the scheduler next chooses v_2 or v_4, respectively.

4.4. Fault at distance 2 from the leader

(a) The parent pointer of the distance-2 neighbor from the leader becomes null

Let us consider Fig. 5. The fault hits the system at v_2. If the scheduler chooses v_2, the system trivially recovers in a single step as R2 can be applied at v_2. If v_3 is chosen by the scheduler, the system has a potential for oscillations; i.e., v_4 or v_2 alternately may become v_3's parent, depending on the x-value of v_2, v_3, and v_4. The system recovers by applying R2 at v_2 or v_4, respectively, when either of them is chosen next by the scheduler.

(b) A new node becomes the parent of the distance-2 faulty node

Consider Fig. 6. If v_1 becomes the new parent of v_2 because of the fault occurring at v_2, then either v_1, v_2, or v_3 can make a move. If v_2 is selected first, R3(a) can be applied now at v_2, as v_3 has a parent if $x(2) \geq x(3)$. If v_3 is selected first, v_3 can select v_2 as its new parent if $x(v_3) \geq x(v_2)$. The recovery is complete following moves by v_2 and v_1 (or v_1 and v_2) (R1) followed by v_4's move (R2), or vice versa. If both v_3 and v_2 are unable to change their parents due to smaller x-values, then recovery is complete by v_1's move first (R1), followed by v_3's move (now R5 can be applied at v_3) and v_4's move (R2).

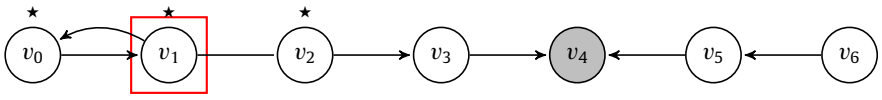

Fig. 8. Fault at distance 3 from the leader. v_0 becomes the new parent of v_1.

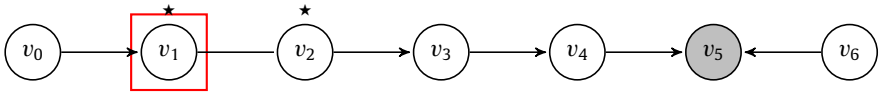

Fig. 9. Fault at distance 4 from the leader. v_1's parent pointer becomes null.

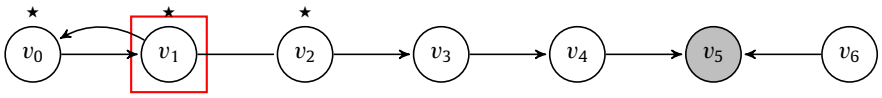

Fig. 10. Fault at distance 4 from the leader. v_0 becomes the new parent of v_1.

4.5. Fault at distance 3 from the leader

(a) The parent pointer of a distance-3 neighbor from the leader becomes null

Consider Fig. 7. If v_2 is chosen to make the first move, then it can alternately choose v_1 or v_3 as its parent for a while, if its x-value is greater than that of both v_1 and v_3. Recovery is completed when v_1 is selected by the scheduler to make a move (R2).

(b) A new node becomes the parent of the distance-3 faulty node

Consider Fig. 8. If v_0 becomes the new parent of v_1 because of the fault occurring at v_1, then the guards of v_0, v_1, and v_2 are true. If v_0 is chosen by the scheduler first, R1 can be applied at v_0. The system recovers when v_1 chooses v_2 as its parent afterwards (R3(a) if applicable) and v_0 makes a move next (R2 can be applied at v_0 possibly after some oscillations of v_1). If v_2 is selected to make a move, and if $x(v_2) \geq x(v_1)$, then v_2 chooses v_1 as its parent and increases its x-value (R3(a)). Now v_3 is still able to change its parent to v_2 regardless of the value of x at v_2 (R5) provided v_0 had made a prior move. Recovery is completed by v_4's move (R2). Otherwise, if v_0 had not made a move prior to v_3, the higher x-value of v_2 will prevent v_3 from changing its parent. The recovery then is completed by v_1's move (R5) followed by v_0's move (R2).

4.6. Fault at distance 4 or beyond from the leader

(a) The parent pointer of a distance-4 neighbor from the leader becomes null

Consider Fig. 9. If v_1 is selected by the scheduler, recovery occurs immediately by R2 or after some oscillations at v_2. Otherwise, if $x(v_2) \geq x(v_1)$, then v_1 becomes v_2's new parent (R4(a)). Similarly v_2 may become v_3's new parent, but this time v_3 sets its x-value higher (R3(a)). Recovery is completed by v_4's move (R5) followed by v_5's move (R2). Note that for fault occurring at distance 4 and beyond from the leader, R5 will no more applicable at the distance-1 neighbor from the leader (v_4 in this case). Therefore, we do not consider hereafter cases beyond a fault at distance 4 from the leader.

(b) A new node becomes the parent of the distance-4 faulty node

Consider Fig. 10. If v_0 becomes the new parent of v_1 because of the fault occurring at v_1, then the guards of v_0, v_1, and v_2 are true. If v_0 is chosen by the scheduler first, R1 can be applied at v_0. The system recovers when v_1 chooses v_2 as its parent afterwards (R3(a) if applicable) and v_0 makes a move next (R2 can be applied at v_0 possibly after some oscillations of v_1). If v_2 is selected to make a move and if $x(v_2) \geq x(v_1)$, then v_2 chooses v_1 as its parent and increases its x-value (R3(a)). Now v_3 cannot apply R5 anymore, and it is unlikely that the fault will propagate beyond v_2. The recovery proceeds through the following steps: v_0 makes a move (R1), after some oscillations at v_2, v_1 chooses v_2 as its parent (R5), and finally v_0 selects v_1 as its parent (R2).

Note that all cases of single-fault configuration beyond distance 4 from the leader will not involve any different recovery steps that are not already covered in the previous scenarios. This is because, even if we shift the original place of the fault away from the leader, its neighborhood that is going to be affected by subsequent recovery steps will remain unchanged. In the recovery mechanism, (R3(b)) and (R4(b)) are not shown, as we only highlighted the moves where the change of parent pointers occurs, leading to the recovery of the system.

4.7. Results

4.7.1. Fault containment in space

Theorem 6. *As $m \to \infty$, the effect of a single failure is restricted within distance 4 of the faulty process on an array, and the contamination number is at most 4 with high probability; i.e., Algorithm 4 is spatially fault containing.*

To prove the result of spatial containment, we need to find out how far the observable variables change from the faulty node. We consider all the subcases of the recovery mechanism.

1. Fault at leader: The fault propagates to at most distance 1.
2. Fault at distance 1 from the leader:
   (a) The parent pointer of the distance-1 neighbor becomes null: The fault propagates to at most distance 1.
   (b) A new node becomes the parent of the distance-1 faulty node: In the recovery steps, we showed that, in Fig. 4, at most v_2's parent or v_4's parent might change. Thus, the fault propagates to at most distance 1.
3. Fault at distance 2 from the leader:
   (a) The parent pointer of the distance-2 neighbor becomes null: Consider Fig. 5. If v_2 is selected, the system recovers immediately. Another possible recovery is through the sequence of moves of v_3 followed by v_4. In the latter case, contamination occurs up to distance 2 from the original faulty node.
   (b) A new node becomes the parent of the distance-2 faulty node: Consider Fig. 6. The worst-case scenario happens when v_3 makes a move and after that v_4 completes the recovery. Contamination occurs up to distance 2 from the original faulty node in this case.
4. Fault at distance 3 from the leader:
   (a) The parent pointer of the distance-3 neighbor becomes null: Consider Fig. 7. The worst-case scenario happens when v_4 has to change its parent. The fault propagates to at most distance 3.
   (b) A new node becomes the parent of the distance-3 faulty node: Consider Fig. 8. The worst-case scenario happens when v_4 has to change its parent. The fault propagates to at most distance 3.
5. Fault at distance 4 from the leader:
   (a) The parent pointer of the distance-4 neighbor becomes null: This is the scenario in which the highest spatial contamination occurs. Consider Fig. 9. In the worst case, a distance-4 node from the faulty node might have to change its parent pointer. In Fig. 9, v_5 is this node.
   (b) A new node becomes the parent of the distance-4 faulty node: Consider Fig. 10. The fault propagates up to distance 1 with high probability. The probability of the fault contaminating beyond distance 1 is $(1 - 1/2^m) \times 1/2^m$, and $\lim_{m\to\infty}(1 - 1/2^m) \times 1/2^m = 0$.

Note that for a fault occurring at distance 4 and beyond from the leader, R5 will no longer be applicable at the distance-1 neighbor from the leader. Therefore, we do not consider the cases beyond a fault at distance 4 from the leader. Hence Algorithm 4 is spatially fault containing, and the highest contamination number is 4 with high probability. □

4.7.2. Fault containment in time

Theorem 7. *The expected number of steps needed to contain a single fault is independent of n, i.e., the number of nodes in the array. Hence Algorithm 4 is fault containing in time.*

Proof. To prove that Algorithm 4 is fault containing in time, we again consider each individual subcase of fault. We show that each subcase can be bounded and that each of them is independent of n, i.e., the size of the array. For each individual case, we calculate the expected number of moves required for recovery. Essentially this means that we are considering the probabilities of the system recovering in a single move, in two moves, in three moves, etc. Adding them up, we get the expectation for each subcase. We denote the number of moves (which is a random variable) as X, and $\Pr X = x$ denotes the probability that the system recovers using x moves.

1. Fault at leader: Consider Fig. 2. This is a trivial case. In this scenario, both v_4 and v_3 have their guards true. Each of them has an equal probability to execute, given a chance. The system recovers in a single move if either of them executes. So the expected number of moves required for recovery is $1 \times 1/2 + 1 \times 1/2 = 1$.
2. Fault at distance 1 from the leader:
   (a) The parent pointer of the faulty node becomes null: Fig. 3. This is again a trivial case. The system recovers in a single move if either v_4 or v_3 executes. The expected number of moves for recovery is the same as before, i.e., $1 \times 1/2 + 1 \times 1/2 = 1$.
   (b) A new node becomes parent of the faulty node: The system can recover following different sequences:
   - v_4, v_2 or v_4, v_3.
   - $v_3, \ldots, v_3$ ($x(4) - x(3)$ times)
   - $v_3, \ldots, v_3$ ($x(4) - x(3) - 1$ times) followed by v_4, v_2 or v_4, v_3
   - v_2, v_4
   - v_2, $v_3, \ldots, v_3$ followed by v_2 or v_4.

   The lengths of the recovery sequences of the first four situations are finite and independent of n, and only the last sequence may be arbitrarily long. We show that its expectation is finite. After v_2 makes a move applying R2, both of v_3's neighbors consider themselves as the leader. Hence, depending on the parent of v_3, each time there are at most two enabled nodes: v_2 and v_3, or v_3 and v_4. Hence, the probability that the scheduler chooses v_3 is 1/2 before recovery completes. Hence, $\Pr$ v_3 is selected consecutively n times $= 1/2^n$. Therefore, if v_2 is selected by the scheduler first, the expected length of the recovery sequence is $2 + \sum_{n=1}^{\infty} n/2^n = 4$.
3. Fault at distance 2 from the leader

(a) The parent pointer of the faulty node becomes null: Regardless of the values of $x(2)$, $x(3)$, and $x(4)$, no matter which nodes the scheduler chooses, there is always a $1/2$ possibility that the system recovers after the selected node makes a move. Hence, the expected recovery time is $\sum_{n=1}^{\infty} n/2^n = 2$.

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be

$$\begin{aligned}\mathbb{E}(X) &= 1 \times \frac{1}{3} + 3\left(\frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{4} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{4} \times \frac{1}{2}\right) + \frac{2}{3}\sum_{n=2}^{\infty} \frac{2n+1}{2^{2n}} \\ &= \frac{151}{108}.\end{aligned}$$

4. Fault at distance 3 from the leader:

(a) The parent pointer of the faulty node becomes null: In this case, the expected number of moves for recovery will be

$$\begin{aligned}\mathbb{E}(X) &= 1 \times \frac{1}{2} + \frac{2}{0} + 3 \times \left(\frac{1}{2^3} + \frac{1}{2^3}\right) + 5 \times \left(4 \times \frac{1}{2^5}\right) + \cdots \\ &= \frac{1}{2} + \sum_{n=1}^{\infty} (2n+1) \times 2n \frac{1}{2^{2n+1}} = \frac{131}{54}.\end{aligned}$$

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be

$$\begin{aligned}\mathbb{E}(X) &= 1 \times \frac{1}{3} + 2 \times 0 + 4 \times \left(\frac{1}{3}\frac{1}{2}\left(1 - \frac{1}{2^m}\right)\frac{1}{2^m}\frac{1}{2}\right) \\ &\quad + 3 \times \left(\frac{1}{3}\frac{1}{2}\frac{1}{2}\right) + \cdots \leq \frac{1}{3} + \frac{25}{9} + 4 \times \frac{1}{3}\frac{1}{2}\frac{1}{4}\frac{1}{2} = \frac{115}{36}.\end{aligned}$$

(The term $\left(1 - \frac{1}{2^m}\right)\frac{1}{2^m}$ is bounded by $1/4$.)

5. Fault at distance 4 from the leader:

(a) The parent pointer becomes null: In this case, the expected number of moves for recovery will be $\mathbb{E}(X) = \frac{1}{2} + \sum_{n=1}^{\infty}(2n+1)\frac{1}{2^{2n+1}} = 10/9$.

(b) A new node becomes parent of the faulty node: In this case, the expected number of moves for recovery will be $\mathbb{E}(X) = 1 \times \frac{1}{3} + 2 \times 0 + 3 \times \left(\frac{1}{3}\frac{1}{2}\frac{1}{2}\right) + 4 \times \frac{1}{3}\frac{1}{4}\frac{1}{2}\frac{1}{2} + 5 \times \frac{1}{3}\frac{1}{2^4} + 7 \times \frac{1}{3}\frac{1}{2^6} + \cdots = 29/27$.

In all individual subcases, the calculation shows that recovery takes place in a finite number of moves and it is independent of n, where n is the size of the array. The maximum expected number of moves required for recovery occurs in case 4(b), i.e., when a fault occurs at distance 3 from the leader and a new node becomes parent of the faulty node. Therefore, Algorithm 4 is fault containing in time. □

Again, there is no overhead in stabilizing the secondary variables. $LC_s = \mathit{true}$ as long as $x(i) \in \mathbb{Z}^+$. So LC holds as soon as LC_p holds. This leads to the following theorem.

Theorem 8. *For single failures, the fault gap equals the containment time.*

5. Conclusions

The major advantage of the proposed techniques is that the fault gap is independent of the network size. This increases the availability of the system by restoring the system's readiness to efficiently tolerate the next single fault within a short time.

The proposed algorithms for the persistent-bit protocol allow the immediate neighbors of the faulty process to be contaminated. As m increases, it becomes increasingly difficult for the effect of the failure to propagate to the distance-2 neighbors and beyond. However, once it propagates to distance-2 neighbors or beyond, the time for recovery increases for higher values of m. The optimal choice of m must balance these two factors.

Our proposed algorithm for the leader election problem on a line topology can easily be extended to a tree network. An array is a special case of a tree network, and for the sake of simplicity, in this paper, we have described the algorithm's behavior in terms of an array. In the case of a general tree topology, one will have to consider all the neighbors of a process i when executing the rules of the algorithm, instead of considering only two neighbors. The expected recovery will thus include Δ, the maximum degree of a node in the tree. Note that such a result also satisfies the definition of weak fault containment [4].

The extension of these algorithms to more general topologies is a topic of future work.

Acknowledgement

The second author's work was supported in part by National Science Foundation grant CNS-0956780.

References

[1] Y. Azar, S. Kutten, B. Patt-Shamir, Distributed error confinement, ACM Trans. Algorithms 6 (3) (2010).
[2] Y. Afek, S. Dolev, Local stabilizer, J. Parallel Distrib. Comput. 62 (5) (2002) 745–765.
[3] S. Kutten, B. Patt-Shamir, Stabilizing time-adaptive protocols, Theoret. Comput. Sci. 220 (1999) 93–111.
[4] A. Dasgupta, S. Ghosh, X. Xiao, Probabilistic fault-containment, in: 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Paris, France, 2007.
[5] A. Dasgupta, S. Ghosh, X. Xiao, Fault containment in weakly stabilizing systems, in: 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Lyon, France, 2009.
[6] S. Devismes, S. Tixeuil, M. Yamashita, Weak vs. Self vs. Probabilistic Stabilization, Distributed Computing Systems, International Conference, vol. 0, 2008. pp. 681–688, Los Alamitos, CA, USA.
[7] S. Kutten, D. Peleg, Fault-local distributed mending, in: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, 1995, pp. 20–27.
[8] S. Ghosh, A. Gupta, T. Herman, S.V. Pemmaraju, Fault-containing self-stabilizing distributed algorithms, in: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, 1996, pp. 45–54.
[9] J. Beauquier, C. Genolini, S. Kutten, Optimal reactive *k*-stabilization: the case of mutual exclusion, in: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, 1999, pp. 209–218.
[10] J. Beauquier, S. Delaet, S. Haddad, A 1-strong self-stabilizing transformer, in: Proceedings of the Eighth Symposium on Self-Stabilizing Systems, 2006.
[11] J.L. Coolidge, The Gambler's ruin, Ann. of Math. 10 (4) (1909) 181–192.
[12] S. Dolev, T. Herman, Superstabilizing protocols for dynamic distributed systems, in: Proceedings of the Second Workshop on Self-Stabilizing Systems, 1995, pp. 3.1–3.15.
[13] S. Ghosh, A. Gupta, S.V. Pemmaraju, Fault-containing network protocols, in: Proceedings of 12th Annual ACM Symposium on Applied Computing, 1997.
[14] M.G. Gouda, The theory of weak stabilization, in: Proceedings of WSS, in: LNCS, vol. 2194, 2001, pp. 114–123.
[15] T. Herman, Superstabilizing mutual exclusion, in: Proceedings of 1st International Conference on Parallel and Distributed Processing: Techniques and Applications, 1995.
[16] S. Ghosh, A. Gupta, An exercise in fault-containment: self-stabilizing leader election, Informat. Process. Lett. 5 (59) (1996) 281–288.
[17] S. Ghosh, A. Gupta, T. Herman, S.V. Pemmaraju, Fault-containing self-stabilizing distributed protocols, Distrib. Comput. (2007).
[18] S. Ghosh, A. Gupta, S.V. Pemmaraju, A fault-containing self-stabilizing algorithm for spanning trees, J. Comput. Informat. 2 (1996) 322–338.
[19] A. Gupta, Fault-containment in self-stabilizing distributed systems, Ph.D. thesis. Department of Computer Science, The University of Iowa, Iowa City, IA, USA, 1997.